

Jour 2 — Git (fin) & Docker

DevOps pour le Développeur Web et Web Mobile

DWWM — ForEach Academy

Formateur : Fabrice Claeys

Programme de la journée

Horaire	Contenu
9h00-9h30	2.5 Rebase — historique linéaire, règle d'or
9h30-9h50	2.6 Versioning — SemVer, tags, releases, CHANGELOG
9h50-10h30	Fin du TP 1 — rebase, tag <code>v1.0.0</code> , release GitHub
10h30-10h45	Mise en place du fil rouge — repo <code>taskflow</code> , <code>main</code> protégée
10h45-11h00	<i>Pause</i>
11h00-11h30	3.1 Machines virtuelles vs conteneurs
11h30-12h30	3.2 Docker Engine et images — premières commandes (démonstration)
13h30-14h15	3.3 Dockerfile — écrire la recette de l'API (démonstration)
14h15-15h00	3.4 Volumes, réseaux, variables d'environnement
15h00-17h00	TP fil rouge — Étape 1 : dockeriser TaskFlow (+ pause) · récap et quiz

Où en est-on ?



- **Hier** : la culture DevOps, Git en équipe (branches, PR, revue, conflits)
- **Ce matin** : finir Git (rebase, versions) → première case **Release**
- **Ensuite** : Docker → la case **Build** : fabriquer une application qui tourne **partout pareil**

Rappel express d'hier

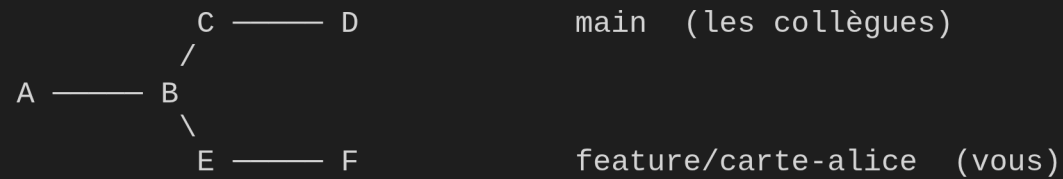
Question	Réponse en une ligne
Une branche, c'est quoi ?	Une étiquette mobile posée sur un commit
Fast-forward ?	<code>main</code> n'avait pas bougé : Git avance juste l'étiquette
Commit de fusion ?	<code>main</code> avait avancé : nouveau commit à deux parents
Une PR ?	Une demande de fusion, relue par un pair et vérifiée par la CI
Un conflit ?	Même ligne modifiée des deux côtés : c'est à vous de choisir

“ Aujourd'hui, on ajoute une deuxième façon d'intégrer `main` dans sa branche : le **rebase**. ”

2.5 Rebase

Le problème : `main` a avancé sans vous

Vous avez créé `feature/carte-alice` lundi. Entre-temps, deux collègues ont fusionné leur travail.



Avant d'ouvrir (ou de fusionner) votre PR, vous devez récupérer `C` et `D`. Deux solutions :

1. **merge** : `git merge main` (vu hier)
2. **rebase** : `git rebase main` (aujourd'hui)

Solution 1 : merge (rappel)



-  Simple, ne réécrit **rien** : sans danger
-  Un commit de fusion **M** à chaque synchronisation
-  Sur un projet actif, l'historique devient un « plat de spaghettis »

```
$ git log --oneline --graph
* 9a1b2c3 Merge branch 'main' into feature/carte-alice
| \
| * 5d6e7f8 feat: ajoute la carte de Bob
```

Solution 2 : rebase — l'idée

Analogie : vous aviez écrit deux paragraphes (E , F) sur la version de lundi d'un document partagé. Le rebase **décolle** vos paragraphes et les **recolle** à la fin de la version du jour, comme si vous aviez commencé ce matin.

AVANT



APRÈS git rebase main



- Vos commits sont **rejoués** un par un au-dessus de `main`
- `E'` et `F'` ont le **même contenu** que `E` et `F`, mais un **nouveau SHA** : ce sont de nouveaux commits

Rebase : les commandes

```
$ git switch feature/carte-alice
$ git fetch origin
$ git rebase origin/main
Successfully rebased and updated refs/heads/feature/carte-alice.
$ git log --oneline --graph
* 3f2a1bc (HEAD -> feature/carte-alice) feat: ajoute le style de la carte d'Alice
* 8e7d6c5 feat: ajoute la carte d'Alice
* 5d6e7f8 (origin/main) feat: ajoute la carte de Bob
* 1a2b3c4 feat: ajoute la carte de Chloé
```

- `git fetch` récupère les nouveautés de GitHub **sans rien modifier** chez vous
- Résultat : un historique **linéaire**, une seule ligne, facile à lire
- Le merge de votre PR dans `main` sera ensuite un simple **fast-forward**

Merge ou rebase ?

git merge main

git rebase main

Historique	Garde la vraie chronologie, avec des bifurcations	Linéaire , comme si tout s'était fait à la suite
Commits existants	Inchangés	Réécrits (nouveaux SHA)
Commit supplémentaire	Oui (commit de fusion)	Non
Conflits	Résolus une fois	Résolus commit par commit
Danger	Aucun	Sur une branche partagée : oui (voir la règle d'or)
Usage typique	Fusionner une PR dans main	Mettre sa branche à jour avant la PR

Un conflit pendant un rebase

```
$ git rebase origin/main
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
error: could not apply 8e7d6c5... feat: ajoute la carte d'Alice
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
```

Même méthode qu'hier, mais on **continue** au lieu de commiter :

```
$ git add index.html                                # après avoir résolu dans VS Code
$ git rebase --continue                             # rejoue le commit suivant
Successfully rebased and updated refs/heads/feature/carte-alice.
```

“

Perdu ? `git rebase --abort` remet tout comme avant le rebase.

”

La règle d'or du rebase

“ *Ne jamais rebaser une branche sur laquelle d'autres personnes travaillent (et jamais `main`).* ”

```
Chez vous, après rebase :   A — B — C — D — E' — F'
Chez Bob (même branche) :  A — B — E — F — G      ← il a construit G sur E et F
```

- Le rebase **remplace** `E`, `F` par `E'`, `F'` : pour Git, ce sont des commits **différents**
- Bob se retrouve avec des doublons et des conflits incompréhensibles
-  Rebaser **votre** branche, que **vous seul** utilisez : aucun problème

Pousser après un rebase

Vos commits ont changé de SHA : GitHub refuse un `git push` normal.

```
$ git push
! [rejected]          feature/carte-alice -> feature/carte-alice (non-fast-forward)
hint: Updates were rejected because the tip of your current branch is behind
$ git push --force-with-lease
+ 7c8d9e0...3f2a1bc feature/carte-alice -> feature/carte-alice (forced update)
```

- `--force-with-lease` : force **seulement si** personne n'a poussé entre-temps sur cette branche
- ⚠ Jamais `--force` tout court, et **jamais** sur `main` (de toute façon, elle est protégée)

`git pull --rebase` : synchroniser sans commit de fusion

`git pull` = `git fetch` + `git merge`. Si vous et un collègue avez commité sur la même branche, vous obtenez un commit « *Merge branch 'main' of github.com...* » inutile.

```
$ git pull --rebase
Successfully rebased and updated refs/heads/main.
```

En faire le comportement par défaut (une seule fois, sur votre poste) :

```
$ git config --global pull.rebase true
```

“ Vos commits **locaux, pas encore poussés** sont rejoués au-dessus de ceux de GitHub : c'est sans danger, ils n'étaient connus que de vous. ”

2.6 Versioning

Pourquoi numéroter les versions ?

“ Le client appelle : « Le bouton supprimer ne marche plus ! » — « Quelle version avez-vous ? » — « Euh... celle de mardi ? » ”

Analogie : les **éditions** d'un livre. « 2^e édition, revue et corrigée » vous dit tout de suite ce qui a changé et laquelle vous avez entre les mains.

Un numéro de version permet de :

- savoir **exactement** ce qui tourne en production
- **revenir** à une version précédente en cas de problème (*rollback*)
- annoncer **l'impact** d'une mise à jour à ceux qui l'utilisent

SemVer : MAJEUR.MINEUR.CORRECTIF

SemVer (*Semantic Versioning*, versionnage sémantique) : chaque nombre a un **sens**.

```
v  2 . 4 . 1
   |   |   |
   |   |   └─ CORRECTIF : correction de bug, rien ne change pour l'utilisateur
   |   └─ MINEUR   : nouvelle fonctionnalité, compatible avec l'existant
   └─ MAJEUR      : changement INCOMPATIBLE (il faudra s'adapter)
```

Changement dans TaskFlow	Avant	Après
Corriger le compteur de tâches	1.3.0	1.3.1
Ajouter le filtre « tâches terminées »	1.3.1	1.4.0
Renommer <code>/api/tasks</code> en <code>/api/v2/todos</code>	1.4.0	2.0.0

“ Quand un nombre augmente, ceux à **sa droite** repassent à 0. Une version `0.x.y` signifie « encore instable ».

”

Les tags Git

Un **tag** est une étiquette **fixe** posée sur un commit (une branche, elle, avance).

	Tag léger	Tag annoté (recommandé)
Commande	<code>git tag v1.0.0</code>	<code>git tag -a v1.0.0 -m "Première version stable"</code>
Contient	Juste un nom	Nom, auteur, date, message
Usage	Repère personnel	Versions publiées

```
$ git tag -a v1.0.0 -m "Première version stable du site promo"
$ git show v1.0.0 --no-patch
tag v1.0.0
Tagger: Alice Diallo <alice@example.com>
Date:   Tue Oct 20 09:42:11 2026 +0200

Première version stable du site promo
```

Lister et pousser les tags

```
$ git tag -l
v0.1.0
v1.0.0
$ git push origin v1.0.0
* [new tag]          v1.0.0 -> v1.0.0
$ git log --oneline -3
3f2a1bc (HEAD -> main, tag: v1.0.0, origin/main) feat: ajoute la carte d'Alice
5d6e7f8 feat: ajoute la carte de Bob
1a2b3c4 feat: ajoute la carte de Chloé
```

- ⚠ `git push` seul **n'envoie pas** les tags : il faut les pousser explicitement
- Revenir voir le code d'une version : `git switch --detach v1.0.0`
- Demain (jour 3), la CI utilisera ces tags pour étiqueter les images Docker

Les releases GitHub

Une **release** = un tag + des **notes de version** lisibles + éventuellement des fichiers à télécharger.

Sur GitHub : **Releases** → **Draft a new release** → choisir le tag `v1.0.0` → **Generate release notes**
→ *Publish release*.

Ou en ligne de commande (outil `gh`, la CLI de GitHub) :

```
$ gh release create v1.0.0 --title "v1.0.0" --generate-notes  
https://github.com/promo-dwwm/site-promo/releases/tag/v1.0.0
```

“Generate release notes liste automatiquement les PR fusionnées depuis la version précédente : encore une raison d'écrire de bons titres de PR.”

Le CHANGELOG

Un fichier `CHANGELOG.md` à la racine : l'historique **pour les humains** (format *Keep a Changelog*).

```
# Changelog

## [1.1.0] - 2026-10-22
### Ajouté
- Filtre « tâches terminées » (#12)
### Corrigé
- Le compteur ne comptait pas les tâches supprimées (#14)

## [1.0.0] - 2026-10-20
- Première version stable
```

“ Le `git log` raconte **comment** le code a changé ; le `CHANGELOG` explique **ce que ça change** pour l'utilisateur. ”

Fin du TP 1 & mise en place du fil rouge

Fin du TP 1 (40 min)

tp/tp1-git-collaboratif/ — suite d'hier

Étape	Commande / geste
1. Créer une nouvelle branche pour un petit changement	<code>git switch -c feature/style-<prenom></code>
2. Attendre qu'un autre membre fusionne une PR dans <code>main</code>	—
3. Rebaser votre branche sur la dernière version de <code>main</code>	<code>git fetch && git rebase origin/main</code>
4. Pousser, ouvrir la PR, la faire relire, fusionner	<code>git push --force-with-lease</code> (si déjà poussée)
5. Un membre tague la version et la pousse	<code>git tag -a v1.0.0 -m "... " && git push origin v1.0.0</code>
6. Créer la release GitHub avec notes générées	<i>Releases → Draft a new release</i>

Critères : `git log --oneline --graph` sur `main` montre un historique lisible ; tag `v1.0.0` visible sur GitHub ; release publiée.

Le fil rouge : votre dépôt `taskflow`

Chacun crée **son** dépôt à partir du starter (`tp/fil-rouge-taskflow/starter/`) :

```
$ cp -r tp/fil-rouge-taskflow/starter ~/taskflow && cd ~/taskflow
$ git init -b main && git add . && git commit -m "chore: initialise TaskFlow"
[main (root-commit) 0a1b2c3] chore: initialise TaskFlow
$ gh repo create taskflow --public --source=. --push
✓ Created repository <vous>/taskflow on GitHub
✓ Pushed commits to https://github.com/<vous>/taskflow.git
```

Vérifiez que le starter fonctionne **sans Docker** :

```
$ cd api && npm ci && npm test
Tests 27 passed | 10 skipped (37)
```

Protéger `main` sur votre dépôt

Settings → Branches → Add branch protection rule → Branch name pattern : `main`

- ☒ *Require a pull request before merging*
- ☐ *Require approvals* : décoché pour l'instant (vous êtes seul pour l'instant ; plus tard, votre binôme relira)
- ☒ *Do not allow bypassing the above settings*

Vérification — ce push **doit** être refusé :

```
$ git commit --allow-empty -m "test: push direct" && git push
remote: error: GH006: Protected branch update failed for refs/heads/main.
! [remote rejected] main -> main (protected branch hook declined)
$ git reset --hard HEAD~1          # on annule ce commit de test
```

“

Dépôt **public** obligatoire avec un compte GitHub gratuit.

”

S3 — Docker

3.1 Machines virtuelles vs conteneurs

Le problème : « ça marche sur ma machine »

TaskFlow, chez trois personnes différentes :

	Poste d'Alice	Poste de Bob	Serveur de prod
Système	Windows 11	macOS 14	Ubuntu 24.04
Node.js	20.11	18.19	20.15
PostgreSQL	16	14	16
Variable DATABASE_URL	définie	oubliée	définie autrement
Résultat	✅ ça marche	❌ erreur au démarrage	? on verra en prod...

Il faut un moyen d'emporter **l'application + tout son environnement**, identique partout.

L'analogie du conteneur maritime

AVANT 1956 : chaque marchandise a sa forme
 tonneaux  caisses  sacs
→ chargement à la main, lent, casse, vols

APRÈS : une boîte standard



même grue, même bateau, même camion

- Le transporteur **ne sait pas** ce qu'il y a dedans... et n'a **pas besoin** de le savoir
- Un **conteneur logiciel** fait pareil : votre application, ses dépendances, sa configuration, dans une boîte **standard** qui tourne de la même façon sur votre PC, dans la CI et sur le serveur
- **Docker** (2013) est l'outil qui a popularisé ces conteneurs

La machine virtuelle (VM)

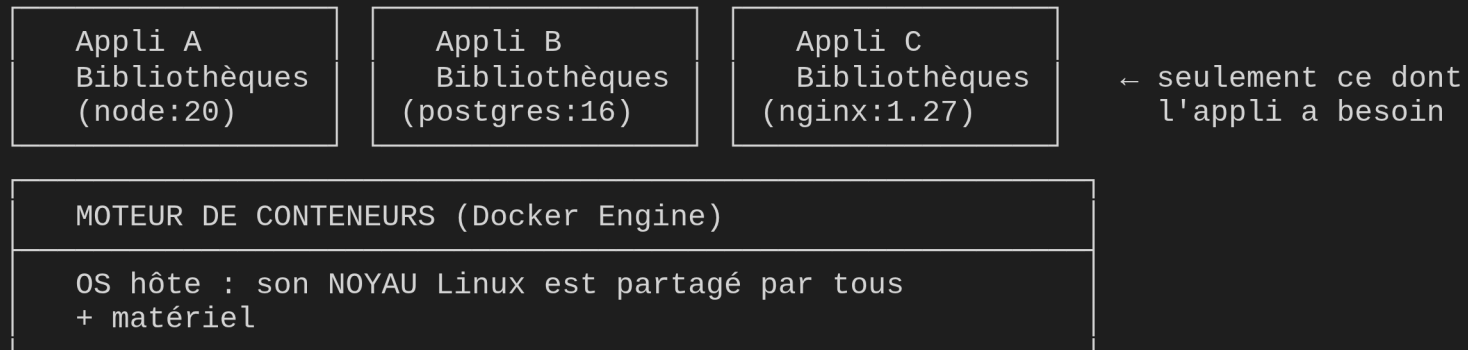
Une **VM** (*Virtual Machine*) simule un ordinateur **complet**, avec son propre système d'exploitation.



- **OS** (*Operating System*) : système d'exploitation • **Hyperviseur** : logiciel qui fait tourner des VM
- Isolation très forte, mais **lourd** : plusieurs Go, démarrage en minutes

Le conteneur

Un **conteneur** est un **processus isolé** qui **partage le noyau** du système hôte.



- **Noyau** (*kernel*) : le cœur du système, qui parle au matériel
- Le conteneur « croit » être seul sur la machine : ses fichiers, son réseau, ses processus
- **Léger** : quelques dizaines de Mo, démarrage en **une seconde**

VM vs conteneur

	Machine virtuelle	Conteneur
Contient	Un OS complet + l'appli	L'appli + ses bibliothèques
Taille typique	Plusieurs Go	Dizaines à centaines de Mo
Démarrage	Minutes	Secondes
Isolation	Très forte (matériel simulé)	Forte (processus isolé, noyau partagé)
Sur une machine	Quelques VM	Des dizaines de conteneurs
Usage typique	Faire tourner un autre OS, un serveur complet	Livrer une application et ses dépendances

Analogie : une VM est une **maison individuelle** (fondations, toiture, chaudière à elle) ; un conteneur est un **appartement** dans un immeuble (murs privés, mais fondations et chaufferie communes).

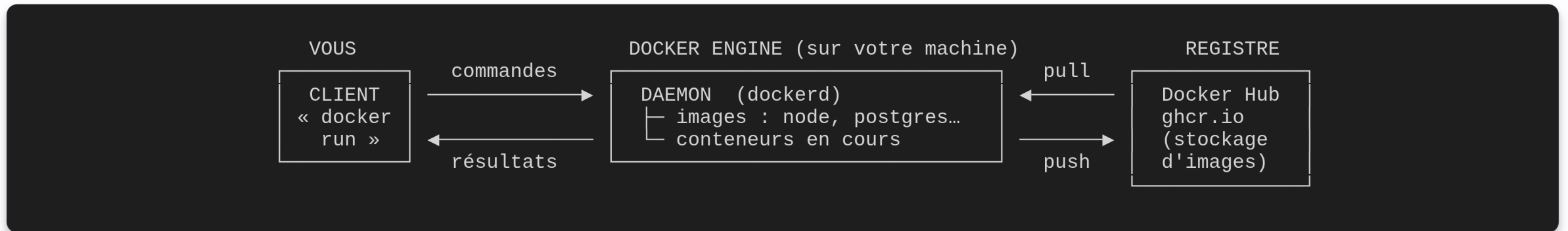
“ Sur Windows et macOS, Docker Desktop fait tourner une petite VM Linux invisible, qui héberge vos conteneurs. ”

Ce que Docker va régler pour TaskFlow

Problème	Avec Docker
« J'ai Node 18, pas 20 »	L'image de l'API embarque Node 20
« Comment j'installe Postgres 16 ? »	<code>docker run postgres:16-alpine</code> — c'est tout
« Ça marche chez moi, pas en prod »	La même image tourne sur le poste, dans la CI et sur le serveur
« Nouvel arrivant : 1 journée d'installation »	Demain : <code>git clone</code> + <code>docker compose up</code> = 5 minutes
« Quelle version est en prod ? »	Image étiquetée <code>taskflow-api:1.4.0</code> ou par SHA de commit

3.2 Docker Engine et images

L'architecture de Docker



- **Client** : la commande `docker` que vous tapez
- **Daemon** : le service en arrière-plan qui fait le vrai travail (construire, lancer, arrêter)
- **Registre** : une « bibliothèque » d'images en ligne ; **Docker Hub** est le registre public par défaut

Vérifier l'installation

```
$ docker version
Client:
 Version:           27.3.1
 OS/Arch:           linux/amd64
 Server: Docker Engine - Community
 Engine:
  Version:          27.3.1
$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

- Les deux parties **Client** et **Server** (le daemon) doivent apparaître
- *Cannot connect to the Docker daemon* → Docker Desktop n'est pas lancé (ou, sous Linux, votre utilisateur n'est pas dans le groupe `docker`)

Image vs conteneur

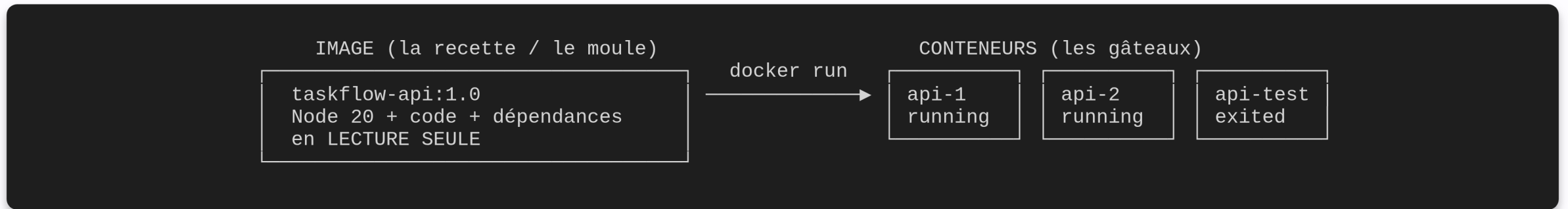


Image	Conteneur
Un modèle figé, en lecture seule	Une instance vivante de ce modèle
Se construit (<code>build</code>) ou se télécharge (<code>pull</code>)	Se lance (<code>run</code>), s'arrête, se supprime
Une seule image...	... autant de conteneurs que l'on veut

“

*En programmation objet : l'image est la **classe**, le conteneur est l'**objet** (`new`).*

”

Une image est faite de couches

Couche conteneur (lecture-écriture)		← créée par docker run, PERDUE à docker rm
COPY src ./src	(40 Ko)	← change à chaque modification du code
RUN npm ci --omit=dev	(25 Mo)	← change si package.json change
COPY package*.json ./	(100 Ko)	
node:20-alpine	(130 Mo)	← partagée avec TOUTES vos images Node

- Chaque couche est **en lecture seule** et **réutilisable** (cache, partage entre images)
- Télécharger une nouvelle version = télécharger **uniquement les couches qui ont changé**

Docker Hub et les tags d'images

Nom complet d'une image : `registre/propriétaire/nom:tag`

Écriture	Signifie
<code>postgres:16-alpine</code>	Image officielle <code>postgres</code> , version 16, basée sur Alpine (Linux minimal, ~5 Mo)
<code>node:20-alpine</code>	Node.js 20 sur Alpine
<code>nginx:1.27-alpine</code>	Serveur web nginx 1.27
<code>ghcr.io/<vous>/taskflow-api:1.0.0</code>	Votre image, sur le registre de GitHub (demain)
<code>nginx</code> (sans tag)	= <code>nginx:latest</code> : ⚠ change dans le temps, à éviter en prod

```
$ docker pull postgres:16-alpine
16-alpine: Pulling from library/postgres
Status: Downloaded newer image for postgres:16-alpine
```

Lancer un conteneur : `docker run`

```
$ docker run -d --name web -p 8080:80 nginx:1.27-alpine
Unable to find image 'nginx:1.27-alpine' locally
1.27-alpine: Pulling from library/nginx
Status: Downloaded newer image for nginx:1.27-alpine
f3c1a9e2b7d4c8e1a0b5d6f7e8c9a0b1c2d3e4f5a6b7c8d9e0f1a2b3c4d5e6f7
```

Option	Rôle
<code>-d</code>	<i>detached</i> : en arrière-plan, vous récupérez la main
<code>--name web</code>	Un nom lisible (sinon Docker en invente un, ex. <code>quirky_turing</code>)
<code>-p 8080:80</code>	Publie un port : <code>8080</code> de votre machine → <code>80</code> du conteneur

Ouvrez <http://localhost:8080> : « *Welcome to nginx!* »

Les ports : `-p HÔTE:CONTENEUR`



- Un conteneur est **isolé** : sans `-p`, son port 80 est **inaccessible** depuis votre navigateur
- **À gauche** : le port de votre machine (vous choisissez, il doit être libre)
- **À droite** : le port sur lequel l'application écoute **dans** le conteneur
- *port is already allocated* → ce port est déjà pris : changez celui de gauche (`-p 8081:80`)

Voir les conteneurs et leurs logs

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  STATUS        PORTS          NAMES
f3c1a9e2b7d4   nginx:1.27-alpine  "/docker-entrypoint...."  Up 2 minutes   0.0.0.0:8080->80/tcp  web
$ docker logs web
/docker-entrypoint.sh: Configuration complete; ready for start up
172.17.0.1 - - [20/Oct/2026:11:42:03 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0..."
$ docker logs -f --tail 20 web          # suivre en direct (Ctrl+C pour sortir)
```

- `docker ps` : les conteneurs **en cours** ; `docker ps -a` : **tous**, y compris arrêtés
- Les logs = ce que l'application écrit sur sa sortie standard (`console.log` en Node)

Entrer dans un conteneur : `docker exec`

```
$ docker exec -it web sh
/ # cat /etc/os-release | head -2
NAME="Alpine Linux"
ID=alpine
/ # ls /usr/share/nginx/html
50x.html    index.html
/ # exit
$ docker exec web nginx -v
nginx version: nginx/1.27.2
```

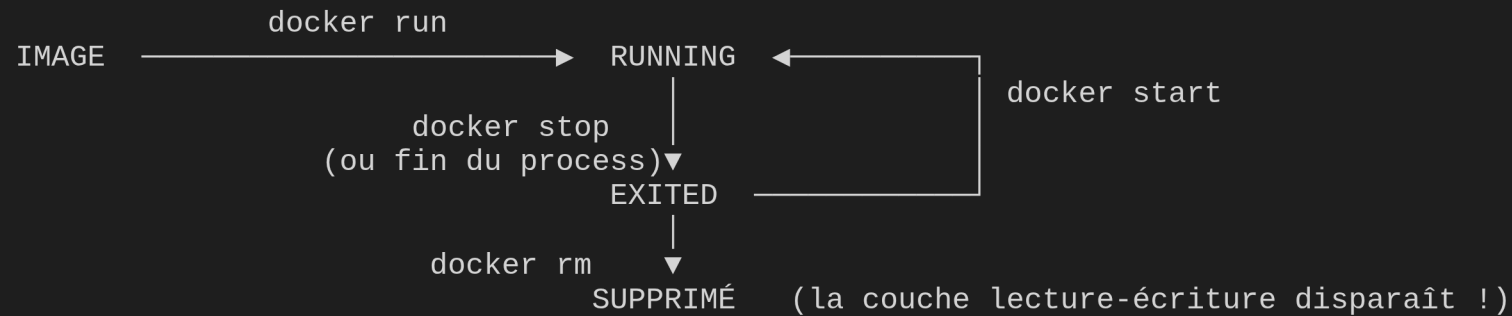
- `exec` lance une commande **dans** un conteneur déjà démarré
- `-it` : mode interactif avec un terminal (pour un shell `sh` ou `bash`)
- Les images Alpine n'ont pas `bash`, seulement `sh`

Arrêter, supprimer, faire le ménage

```
$ docker stop web
web
$ docker ps -a
CONTAINER ID   IMAGE          STATUS              NAMES
f3c1a9e2b7d4   nginx:1.27-alpine Exited (0) 5 seconds ago   web
$ docker rm web
web
$ docker images
REPOSITORY    TAG          IMAGE ID          CREATED          SIZE
nginx         1.27-alpine  1ae23480369f     3 weeks ago     47.9MB
```

- Un conteneur arrêté **existe encore** (on peut le relancer avec `docker start web`)
- `docker rm -f web` : arrête **et** supprime en une fois
- `docker rmi nginx:1.27-alpine` supprime l'image ; `docker system prune` fait le ménage global

Le cycle de vie d'un conteneur



“ Un conteneur vit **tant que son processus principal tourne**. Si l'API plante, le conteneur passe en **Exited** : `docker logs` vous dira pourquoi. ”

Antisèche des commandes de base

Commande	Effet
<code>docker pull image:tag</code>	Télécharge une image
<code>docker run -d --name n -p 8080:80 image</code>	Crée et démarre un conteneur
<code>docker ps</code> / <code>docker ps -a</code>	Liste les conteneurs (actifs / tous)
<code>docker logs -f n</code>	Affiche (et suit) les logs
<code>docker exec -it n sh</code>	Ouvre un shell dans le conteneur
<code>docker stop n</code> / <code>docker start n</code>	Arrête / redémarre
<code>docker rm n</code> / <code>docker rm -f n</code>	Supprime (forcé : même s'il tourne)
<code>docker images</code> / <code>docker rmi image</code>	Liste / supprime les images

Démo — Premiers pas avec Docker

demos/demo-docker-premiers-pas/

1. `docker run hello-world` : que s'est-il passé ? (pull → création → exécution → fin)
2. Un nginx sur le port 8080, consulté dans le navigateur
3. `docker ps` , `docker logs -f` pendant qu'on rafraîchit la page
4. `docker exec` : modifier `index.html` **dans** le conteneur... puis `docker rm` : la modification est **perdue**
5. Deux conteneurs depuis la même image, sur deux ports différents



*Question pour la suite : comment mettre **notre** code dans une image ?*



3.3 Dockerfile

Pourquoi un Dockerfile ?

Un **Dockerfile** est la **recette écrite** de votre image : un fichier texte, versionné dans Git, qui décrit étape par étape comment la fabriquer.

Sans Dockerfile	Avec Dockerfile
« Installe Node 20, puis lance npm ci, puis... » (dans un README, ou dans la tête de quelqu'un)	Une recette exécutable par n'importe qui, n'importe où
Chaque poste est différent	<code>docker build</code> produit la même image partout
Modifier l'environnement = prévenir tout le monde	Modifier l'environnement = une PR relue

“ Le Dockerfile est l'une des **preuves** évaluées en fin de formation (critère « Dockerfile fonctionnel »). ”

Les instructions essentielles

Instruction	Rôle	Exemple
FROM	Image de départ	FROM node:20-alpine
WORKDIR	Dossier de travail (créé si besoin)	WORKDIR /app
COPY	Copie des fichiers de votre projet vers l'image	COPY package*.json ./
RUN	Exécute une commande pendant la construction	RUN npm ci --omit=dev
ENV	Variable d'environnement par défaut	ENV PORT=3000
EXPOSE	Documente le port écouté (ne publie rien)	EXPOSE 3000
USER	Utilisateur qui lance l'appli (pas root)	USER node
CMD	Commande lancée au démarrage du conteneur	CMD ["node", "src/server.js"]

RUN = à la *cuisson* de l'image (une fois) ; **CMD** = à chaque *démarrage* d'un conteneur.

Le Dockerfile de l'API TaskFlow

```
# api/Dockerfile
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY src ./src
ENV NODE_ENV=production PORT=3000
EXPOSE 3000
USER node
HEALTHCHECK CMD wget -qO- http://localhost:3000/health || exit 1
CMD ["node", "src/server.js"]
```

- `npm ci --omit=dev` : installe exactement le `package-lock.json`, **sans** les outils de dev (Vitest, ESLint)
- `USER node` : l'utilisateur non-administrateur fourni par l'image Node — si l'appli est piratée, l'attaquant n'est pas root
- `HEALTHCHECK` : Docker interroge `/health` régulièrement et affiche `healthy` / `unhealthy`

Construire l'image : `docker build`

```
$ cd api
$ docker build -t taskflow-api:dev .
[+] Building 14.2s (10/10) FINISHED
=> [internal] load build definition from Dockerfile 0.0s
=> [1/5] FROM docker.io/library/node:20-alpine 4.1s
=> [2/5] WORKDIR /app 0.1s
=> [3/5] COPY package*.json ./ 0.0s
=> [4/5] RUN npm ci --omit=dev 8.7s
=> [5/5] COPY src ./src 0.0s
=> => naming to docker.io/library/taskflow-api:dev 0.0s
```

- `-t taskflow-api:dev` : le nom (*tag*) de l'image
- Le `.` final : le **contexte** = le dossier envoyé à Docker (d'où `COPY` lit les fichiers)

Lancer l'API en conteneur

```
$ docker run -d --name api -p 3000:3000 taskflow-api:dev
$ docker ps -a --format 'table {{.Names}}\t{{.Status}}'
NAMES      STATUS
api        Exited (1) 3 seconds ago
$ docker logs api
Error: connect ECONNREFUSED 127.0.0.1:5432
```

-  L'image se construit et le conteneur **démarre** : Node 20 et le code sont bien dedans
-  Au démarrage, l'API se connecte à PostgreSQL pour créer la table... et il n'y a pas de base
- Et même avec une base : dans un conteneur, `localhost` (`127.0.0.1`) désigne **le conteneur lui-même**
- On règle ça en 3.4 avec un **réseau** Docker et la variable `DATABASE_URL`

L'ordre des instructions et le cache

Docker **réutilise** une couche si l'instruction et ses fichiers n'ont pas changé. Dès qu'une couche change, **toutes les suivantes** sont reconstruites.

❌ Mauvais ordre

COPY . .

RUN npm ci

← change souvent

← refait à CHAQUE
modification

✅ Bon ordre

COPY package*.json ./

RUN npm ci

COPY src ./src

← change rarement

← en cache 👍

← seule couche refaite

```
$ docker build -t taskflow-api:dev . # après avoir modifié src/app.js
=> CACHED [3/5] COPY package*.json ./ 0.0s
=> CACHED [4/5] RUN npm ci --omit=dev 0.0s
=> [5/5] COPY src ./src 0.0s
```

“

Règle : ce qui change **rarement** en haut, ce qui change **souvent** en bas. 14 s → 1 s.

”

`.dockerignore` : ne pas tout envoyer

Comme `.gitignore`, mais pour le **contexte** de build :

```
# api/.dockerignore
node_modules
npm-debug.log
.env
tests
.git
```

- `node_modules` : réinstallé **dans** l'image (et celui de votre poste peut être compilé pour Windows/macOS)
- `.env` : vos secrets locaux ne doivent **jamais** finir dans une image
- Contexte plus petit = build plus **rapide**

```
$ docker build -t taskflow-api:dev .
=> [internal] load build context           transferring context: 21.3kB   ← au lieu de 48 Mo
```

Le multi-stage : un aperçu avec le front

Le front Vite doit être **compilé** (Node), puis **servi** (nginx). Node n'est plus utile une fois `dist/` produit.

```
# front/Dockerfile
FROM node:20-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

FROM nginx:1.27-alpine
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/dist /usr/share/nginx/html
EXPOSE 80
```

Analogie : on cuisine dans la cuisine (étape `build`), mais on ne sert au client **que l'assiette** (étape nginx), pas les casseroles.

Pourquoi le multi-stage ?

```
$ docker images
REPOSITORY          TAG          SIZE
taskflow-front       single       412MB
taskflow-front       dev          49MB
```

← Node + node_modules + sources + dist
← nginx + dist uniquement

	Une seule étape	Multi-stage
Taille	~400 Mo	~50 Mo
Contient les sources et outils de build	Oui	Non
Surface d'attaque	Grande	Petite
Temps de téléchargement sur le serveur	Long	Court

“ Seule la **dernière** étape compose l'image finale. `COPY --from=build` va chercher des fichiers dans une étape précédente. ”

La configuration nginx du front

```
# front/nginx.conf
server {
    listen 80;
    root /usr/share/nginx/html;

    location /api/ { proxy_pass http://api:3000; }
    location /health { proxy_pass http://api:3000; }

    location / { try_files $uri /index.html; }
}
```

- nginx sert les fichiers de `dist/` (HTML, JS, CSS)
- Les requêtes `/api/...` sont **relayées** (*proxy*) vers le conteneur nommé `api`, port 3000
- `api` est un **nom de conteneur** : ça ne marchera que sur un **réseau Docker** commun (3.4)

Bonnes pratiques Dockerfile

✓ Faire

✗ Éviter

Image de base précise et légère : <code>node:20-alpine</code>	<code>FROM node</code> (= <code>latest</code> , change sans prévenir)
<code>COPY package*.json</code> + <code>npm ci</code> avant <code>COPY src</code>	<code>COPY . .</code> en premier
<code>.dockerignore</code> avec <code>node_modules</code> , <code>.env</code> , <code>.git</code>	Envoyer tout le dossier
<code>USER node</code> (non-root)	Tout lancer en root
<code>npm ci --omit=dev</code>	<code>npm install</code> (versions non garanties)
Multi-stage pour ce qui se compile	Garder les outils de build dans l'image finale
Config par variables d'environnement	Mot de passe écrit dans le Dockerfile

Démo — Écrire un Dockerfile

demos/demo-dockerfile/

1. Une mini-appli Express : écrire le Dockerfile **ligne par ligne**, en expliquant chaque instruction
2. `docker build`, puis `docker run -p` : ça répond
3. Modifier une ligne de code, reconstruire : observer les lignes `CACHED`
4. Inverser l'ordre des `COPY` : observer que `npm ci` est refait à chaque fois
5. `docker history` : retrouver les couches et leur taille

“

À vous : `exercices/jour2/` contient un **Dockerfile à corriger** (5 erreurs cachées).

”

3.4 Volumes, réseaux, variables d'environnement

Le problème : les données sont éphémères

```
$ docker run -d --name db -e POSTGRES_PASSWORD=taskflow postgres:16-alpine
$ docker exec db psql -U postgres -c "CREATE TABLE tasks(title text);"
CREATE TABLE
$ docker exec db psql -U postgres -c "INSERT INTO tasks VALUES ('Acheter du pain');"
INSERT 0 1
$ docker rm -f db && docker run -d --name db -e POSTGRES_PASSWORD=taskflow postgres:16-alpine
$ docker exec db psql -U postgres -c "SELECT * FROM tasks;"
ERROR:  relation "tasks" does not exist
```

Les données étaient dans la **couche lecture-écriture** du conteneur : `docker rm` l'a **supprimée**.

“ **Analogie** : un conteneur est une chambre d'hôtel. Quand vous la rendez, le ménage efface tout. Vos affaires importantes vont dans un **casier** à part : le **volume**. ”

Trois façons de stocker des données

CONTENEUR db

/tmp/... → couche du conteneur

← éphémère : perdu à docker rm

/var/lib/postgresql/data

→ VOLUME NOMMÉ "taskflow-db-data"
géré par Docker, survit au conteneur

/app/src

→ BIND MOUNT ./api/src
un dossier de VOTRE machine

	Couche du conteneur	Volume nommé	Bind mount
Survit à <code>docker rm</code>	✗	✓	✓
Géré par	Docker	Docker	Vous (un dossier du projet)
Usage	Fichiers temporaires	Bases de données	Développement (code modifié en direct)

Le volume nommé en pratique

```
$ docker volume create taskflow-db-data
$ docker run -d --name db -e POSTGRES_PASSWORD=taskflow \
  -v taskflow-db-data:/var/lib/postgresql/data postgres:16-alpine
$ docker exec db psql -U postgres -c "CREATE TABLE tasks(title text);"
$ docker exec db psql -U postgres -c "INSERT INTO tasks VALUES ('Acheter du pain');"
$ docker rm -f db                                     # puis on relance EXACTEMENT la même commande run
$ docker run -d --name db -e POSTGRES_PASSWORD=taskflow \
  -v taskflow-db-data:/var/lib/postgresql/data postgres:16-alpine
$ docker exec db psql -U postgres -c "SELECT * FROM tasks;"
  title
-----
Acheter du pain
```

`-v NOM_DU_VOLUME:CHEMIN_DANS_LE_CONTENEUR` · `docker volume ls` pour les lister

Le bind mount : pour développer

```
$ docker run -d --name api-dev -p 3000:3000 \
  -v "$(pwd)/src:/app/src" \
  taskflow-api:dev node --watch src/server.js
$ docker logs -f api-dev
Restarting 'src/server.js'           ← après avoir sauvegardé app.js dans VS Code
```

- Le dossier `src` de **votre machine** apparaît dans le conteneur : vos modifications sont **immédiatement** visibles
- Pratique en développement, **jamais** en production (on y déploie une image figée)
- Le chemin de gauche commence par `/` ou `./` (un chemin) ; sinon c'est un **nom de volume**

“ ⚠ *Supprimer un volume supprime les données : `docker volume rm` et `docker compose down -v` sont **définitifs**.* ”

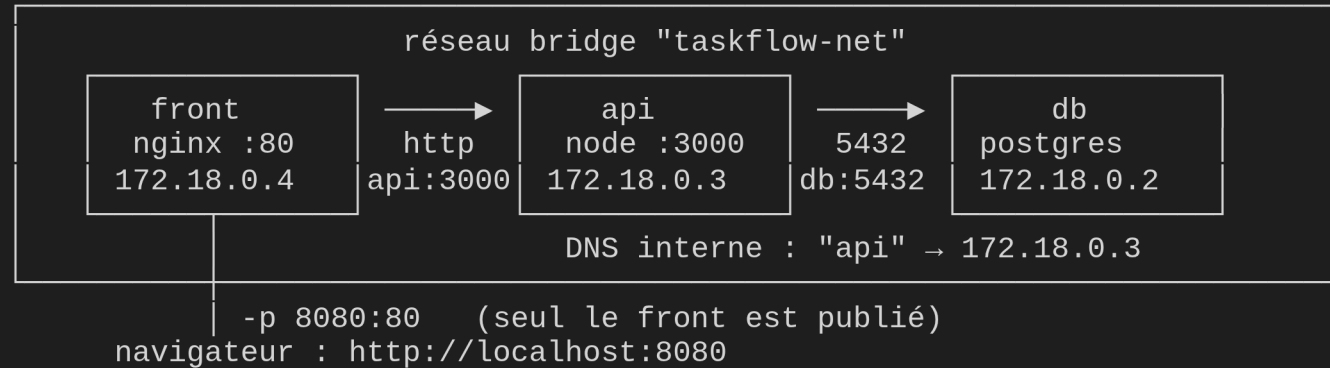
Le problème : localhost dans un conteneur



- Chaque conteneur a **son propre réseau** : localhost désigne le conteneur **lui-même**
- Il faut que api puisse joindre db ... **par son nom**

Le réseau bridge et la résolution par nom

Votre machine



Sur un réseau **créé par vous**, Docker fournit un **DNS** (annuaire) : chaque conteneur est joignable par **son nom**.

Les réseaux en pratique

```
$ docker network create taskflow-net
$ docker run -d --name db --network taskflow-net \
    -e POSTGRES_USER=taskflow -e POSTGRES_PASSWORD=taskflow -e POSTGRES_DB=taskflow \
    -v taskflow-db-data:/var/lib/postgresql/data postgres:16-alpine
$ docker run --rm --network taskflow-net alpine ping -c 1 db
PING db (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.112 ms
$ docker network inspect taskflow-net --format '{{range .Containers}}{{.Name}} {{end}}'
db
```

- `--network taskflow-net` : branche le conteneur sur ce réseau
- `--rm` : supprime le conteneur dès qu'il s'arrête (pratique pour un test)
- ⚠ Le réseau `bridge` **par défaut** (sans `--network`) ne résout **pas** les noms

Variables d'environnement : `-e` et `--env-file`

Même image, configuration différente selon l'environnement : on passe la config **au lancement**.

```
$ docker run -d --name api --network taskflow-net -p 3000:3000 \  
  -e DATABASE_URL=postgres://taskflow:taskflow@db:5432/taskflow \  
  -e APP_ENV=local \  
  taskflow-api:dev
```

Ou depuis un fichier (jamais commité, il est dans `.gitignore`) :

```
$ cat api/.env  
DATABASE_URL=postgres://taskflow:taskflow@db:5432/taskflow  
APP_ENV=local  
$ docker run -d --name api --network taskflow-net -p 3000:3000 --env-file api/.env taskflow-api:dev
```

Ne jamais mettre un secret dans une image

```
ENV DATABASE_URL=postgres://taskflow:MotDePasseSuperSecret@db:5432/taskflow # ❌ NON !
```

```
$ docker history taskflow-api:dev --format '{{.CreatedBy}}' | grep ENV  
ENV DATABASE_URL=postgres://taskflow:MotDePasseSuperSecret@db:5432/taskflow
```

- Une image se **partage** (registre, collègues, serveur) : tout ce qu'elle contient est **lisible**
- Même si une couche suivante « supprime » un fichier, il reste dans la couche précédente
- ☒ Secrets passés **au lancement** (`-e`, `--env-file`) ; demain : secrets GitHub Actions ; jour 4 : `.env` de la prod en `chmod 600`

“

`.env` dans `.gitignore` et dans `.dockerignore`. Toujours.

”

Tout assembler : db + api sur le même réseau

```
$ docker rm -f api
$ docker run -d --name api --network taskflow-net -p 3000:3000 \
  --env-file api/.env taskflow-api:dev
$ curl -s http://localhost:3000/health
{"status":"ok","db":"up","version":"dev","env":"local"}
$ curl -s -X POST http://localhost:3000/api/tasks \
  -H "Content-Type: application/json" -d '{"title":"Dockeriser TaskFlow"}'
{"id":1,"title":"Dockeriser TaskFlow","done":false,"created_at":"2026-10-20T14:58:12.401Z"}
$ docker ps --format 'table {{.Names}}\t{{.Status}}'
NAMES      STATUS
api        Up 30 seconds (healthy)
db         Up 12 minutes
```

db: "up" 🎉 — C'est exactement ce que vous allez faire, puis compléter avec le front, dans le TP.

TP fil rouge — Étape 1 : Dockeriser TaskFlow

`tp/fil-rouge-taskflow/etape1-docker.md`

Étape 1 — Déroulé (2h)

#	Tâche	Résultat attendu
1	Créer le réseau <code>taskflow-net</code> et le volume <code>taskflow-db-data</code>	<code>docker network ls</code> , <code>docker volume ls</code>
2	Lancer Postgres 16 (<code>POSTGRES_USER/PASSWORD/DB</code> , volume, réseau)	<code>docker logs db</code> : <i>ready to accept connections</i>
3	Écrire <code>api/Dockerfile</code> + <code>api/.dockerignore</code> , construire	<code>docker build -t taskflow-api:dev api</code> réussit
4	Lancer l'API sur le réseau avec <code>--env-file</code>	<code>curl localhost:3000/health</code> → <code>"db":"up"</code>
5	Écrire <code>front/Dockerfile</code> multi-stage + <code>front/nginx.conf</code>	Image <code>taskflow-front:dev</code> < 60 Mo
6	Lancer le front (<code>-p 8080:80</code>) sur le réseau	http://localhost:8080 : créer une tâche fonctionne
7	<code>docker rm -f db</code> , relancer <code>db</code> avec le même volume	La tâche créée est toujours là
8	Commiter sur une branche, ouvrir une PR , fusionner	<code>main</code> contient les Dockerfiles

Étape 1 — Vérifications finales

```
$ docker ps --format 'table {{.Names}}\t{{.Image}}\t{{.Status}}\t{{.Ports}}'
NAMES      IMAGE                STATUS              PORTS
front      taskflow-front:dev   Up 2 minutes        0.0.0.0:8080->80/tcp
api        taskflow-api:dev     Up 5 minutes (healthy) 0.0.0.0:3000->3000/tcp
db         postgres:16-alpine   Up 9 minutes        5432/tcp
$ curl -s http://localhost:8080/health
{"status":"ok","db":"up","version":"dev","env":"local"}
$ docker exec db psql -U taskflow -c "SELECT count(*) FROM tasks;"
count
-----
      3
```

Questions de réflexion (après que ça marche) : pourquoi le port 5432 n'est-il **pas** publié ? Que se passerait-il sans volume ? Combien de commandes `docker run` avez-vous tapées... ?

Étape 1 — Si ça ne marche pas

Symptôme	Cause probable	Piste
port is already allocated	Port déjà utilisé sur votre machine	Changer le port de gauche : <code>-p 8081:80</code>
<code>ECONNREFUSED</code> <code>127.0.0.1:5432</code>	<code>localhost</code> dans <code>DATABASE_URL</code>	Utiliser l'hôte <code>db</code> dans l'URL
<code>ENOTFOUND db</code>	<code>api</code> et <code>db</code> sur des réseaux différents	<code>--network taskflow-net</code> sur les deux
Conteneur en <code>Exited (1)</code>	L'application a planté au démarrage	<code>docker logs api</code>
<code>502 Bad Gateway</code> sur <code>/api</code>	nginx ne trouve pas le conteneur <code>api</code>	Nom exact <code>api</code> + même réseau
<code>npm ci</code> échoue au build	<code>package-lock.json</code> absent du contexte	Vérifier <code>COPY package*.json ./</code> et <code>.dockerignore</code>
La tâche a disparu	Volume oublié ou mal nommé	<code>docker volume ls</code> , même <code>-v</code> aux deux lancements

Exercices complémentaires : `exercices/jour2/` (`rebase`, `Dockerfile` à corriger, `quiz images / conteneurs`).

Récap de la journée

Bloc	L'essentiel
2.5 Rebase	Rejoue vos commits au-dessus de <code>main</code> → historique linéaire ; jamais sur une branche partagée
2.6 Versioning	SemVer <code>MAJEUR.MINEUR.CORRECTIF</code> ; tag annoté + <code>git push origin <tag></code> ; release ; CHANGELOG
3.1 VM vs conteneur	VM = OS complet (maison) ; conteneur = processus isolé, noyau partagé (appartement)
3.2 Images	Image = recette en couches, lecture seule ; conteneur = instance ; <code>run</code> , <code>ps</code> , <code>logs</code> , <code>exec</code> , <code>rm</code>
3.3 Dockerfile	<code>FROM</code> → <code>CMD</code> ; ce qui change peu en haut (cache) ; <code>.dockerignore</code> ; multi-stage
3.4 Données & réseau	Volume nommé pour la BDD ; réseau créé = DNS par nom ; config par <code>-e</code> / <code>--env-file</code>

Demain : remplacer toutes ces commandes `docker run` par **un seul fichier** (Docker Compose), puis automatiser les vérifications avec **GitHub Actions**.

Quiz — 5 questions

1. Pourquoi ne faut-il **jamais** rebaser une branche partagée ?
2. TaskFlow est en `2.3.4`. Vous ajoutez l'export des tâches en CSV, sans rien casser. Nouvelle version ?
3. Quelle est la différence entre une **image** et un **conteneur** ?
4. Pourquoi copier `package*.json` et lancer `npm ci` **avant** de copier `src/` ?
5. Dans le conteneur `api`, pourquoi `DATABASE_URL` utilise-t-il l'hôte `db` et non `localhost` ?



Réfléchissez 2 minutes, notez vos réponses, puis passez à la slide suivante.



Quiz — Réponses

1. Le rebase **réécrit** les commits (nouveaux SHA) : les collègues qui ont construit sur les anciens se retrouvent avec doublons et conflits
2. **2.4.0** : nouvelle fonctionnalité compatible → MINEUR + 1, CORRECTIF remis à 0
3. L'**image** est le modèle figé en lecture seule (la recette) ; le **conteneur** en est une instance qui tourne (le gâteau)
4. Pour profiter du **cache** : tant que les dépendances ne changent pas, la couche `npm ci` est réutilisée et le build prend 1 s au lieu de 15
5. Dans un conteneur, `localhost` = **le conteneur lui-même** ; sur un réseau Docker créé, chaque conteneur est joignable **par son nom** (DNS interne)

Glossaire du jour (1/2)

Terme	Définition
Rebase	Rejouer ses commits au-dessus d'une autre branche (nouveaux SHA, historique linéaire)
--force-with-lease	Push forcé qui refuse d'écraser le travail poussé par quelqu'un d'autre
SemVer	<i>Semantic Versioning</i> : MAJEUR.MINEUR.CORRECTIF
Tag	Étiquette fixe sur un commit, souvent un numéro de version (v1.0.0)
Release	Sur GitHub : un tag + des notes de version (+ fichiers)
CHANGELOG	Fichier qui résume, pour les humains, ce qui change à chaque version
Rollback	Retour à une version précédente après un problème
VM	<i>Virtual Machine</i> : ordinateur simulé avec son propre système d'exploitation
Hyperviseur	Logiciel qui fait tourner des VM (VirtualBox, VMware, Hyper-V, KVM)
Noyau (kernel)	Cœur du système d'exploitation ; partagé par les conteneurs

Glossaire du jour (2/2)

Terme	Définition
Conteneur	Processus isolé lancé à partir d'une image
Image	Modèle en lecture seule, en couches, à partir duquel on lance des conteneurs
Couche (<i>layer</i>)	Résultat d'une instruction du Dockerfile, mise en cache et partagée
Daemon (<code>dockerd</code>)	Service en arrière-plan qui construit et exécute les conteneurs
Registre	Stockage d'images en ligne (Docker Hub, ghcr.io)
Dockerfile	Recette texte pour construire une image
Multi-stage	Dockerfile en plusieurs étapes ; seule la dernière forme l'image finale
Volume	Stockage géré par Docker, qui survit à la suppression du conteneur
Bind mount	Dossier de votre machine monté dans un conteneur
Réseau bridge	Réseau virtuel entre conteneurs ; créé par vous, il résout les noms (DNS)

Demain

Docker Compose : tout TaskFlow en **une commande**

GitHub Actions : chaque PR vérifiée automatiquement

Pensez à pousser votre travail : `git status` doit être propre ce soir

Questions ?